Introductory Programming 2021 Final Project: Searchly

Group 18
Ryan Williams, Lukas Dannebrog Jensen, Bence Kovac and Ausrine Kriucokaite
{rywi,lukj,bkov,aukr}@itu.dk

February 13, 2022

Introduction

This document reports on the search engine project that we developed during the Introductory Programming course at the IT University of Copenhagen. In Sections 1–5 we will report on our solutions to the mandatory tasks posed in the project description. The description for each solution is roughly split up into the following parts:

- **Task:** A short review on the task that we had to solve.
- **Approach:** An informal, high-level description of how we solved the task.
- **Solution:** A detailed technical description of our solution to the task.
- **Reflections:** Thoughts about solution, including known bugs or shortcomings.

The source code of our project is handed in as a single zip file called `group18.zip`. The directory `GP21-group18` contains the project-files that solves the mandatory tasks. All the code was written by us or contained in the template, some of the hashcode methods were generated were generated by VS-code's source action command. Our code is also available on ITU's Github: `https://github.itu.dk/mahv/GP21-group18`.

Statement of Contribution

Given the experience and required knowledge for this task, we have decided to establish a team lead position who would be responsible for the overall architecture and who would act as a lead developer in case any issues occurred. The rest of the team focused on more hands-on experience while coding the solution. We have decided to split the mandatory tasks based on the level of individual experience, so that everyone was equally challenged and gained the most out of this project.

Lukas took on the team lead role and guided team members through the assigned tasks. Bence took the lead in implementing Task 4 - Refined Queries and Task 5 - Ranking Algorithm together with Lukas. Ryan and Ausrine took the lead in implementing Task 3 - Inverted Index and Task 2 - Modifying the Basic Search Engine together with Lukas. Refactoring phase was completed as a whole group in order to make sure that everyone got familiar with the code. It is important to note that everyone participated in group discussions and contributed to developing solutions for every task as well as committed part of the code for this project to the repository.

Task 1: Refactoring

Task The given task was to refactor the existing code to establish a cleaner code base and prepare it for the implementation of new features.

Approach We have decided to introduce new classes for our initial prototype to achieve higher cohesiveness, less coupling and reduce overall complexity within the system. In addition, we have decided to extract parts of the code into separate methods, rename variables and discarded the use of local variable type inference (`var`), where the data type was not obvious, to improve code readability.

Solution We have gone through several iterations of the class diagram. We started out with a class diagram that could support task 2, then gradually updated it to support the rest of the required functionality. The final version of initial prototype can be seen in figure 1.

Solution We decided to break down the system into separate parts that could be represented as an individual object with it's own behaviour and attributes. Also, we wanted to make sure that the new architecture could support new features developed in tasks 2-5. Therefore, we have introduced 5 additional classes that all combined create an improved version of the search engine:

- `Page.java` - represents a `Page` object. It is responsible for storing all the given information in the data files: a title, an url and content, as well as modifying the title and url fields into a JSON format.
- `WebSet.java` - represents a set of `Pages`. It is responsible for enabling search engine to support more complex search queries by creating intersections and unions between sets, as well as unifying multiple websets.
- `Ranker.java` - represents a `Ranker` object. It is responsible for ranking matched queries by utilizing a term frequency-inverse document score (tf-idf).
- `QueryHandler.java` - represents a `QueryHandler` object. It is responsible for the processing of search queries and matching them with the pages by utilizing the Inverted Index.
- `InvertedIndex.java` - represents data structure based on the inverted index. It is responsible for storing a mapping of search queries and corresponding pages.

- `WebServer.java` - represents a `WebServer` object. It is responsible for creating, from file, processing and delivering web pages, it also takes the http request, that will trigger the search.

In regards of variables, instead of using "var" for locally declared variables, we have updated them into data types. For example, "`var filename`" was refactored to "`String filename`". We have applied this approach to the `WebServer` class. We have observed that most of the methods in the same class were responsible for several functions that could be refactored further into individual methods. We have extracted code lines that create a HTTP content into one method called `createAllContent()`. Code related to the terminal message that outputs access to a localhost was combined into a `displayLocalhost(int port)` method. Furthermore, we have observed that some of the functionality should not be handled in the `WebServer` class as it falls out of a webserver scope. Therefore we have placed all the functionality that relates to reading, loading and searching pages to the `QueryHandler` and `InvertedIndex` classes, allowing us to use query handler object instead. This approach was implemented into refactoring of `search(HttpExchange io)` method. We have tested the refactored code using unit test cases provided in the `WebServerTest`.java file and implemented java docs for every method.

Reflection The potential shortcoming of our refactoring and initial prototype is that it does not fully utilize the main four principles of object-oriented programming. We have focused on encapsulation by keeping fields and methods private and polymorphism by overriding methods, however we are aware that it lacks inheritance and abstraction. Given the additional time, we would utilize interfaces and abstract classes for the data structures, we had thoughts on making an interface called StructuredWeb, that would be implemented by InvertedIndex and other potential data structures. We have also tried to refactor some of the methods into streams, however later on we have discovered that this might not be a correct approach based on the overall team experience. However, given the additional time and more in-depth knowledge, we would refactor methods into streams where applicable.

Task 2: Modifying the Basic Search Engine

Task The given task was to output a message informing a user that no web pages matched the given search term if no matches were found. Additionally, the program could create a web page only if it contained a title and at least one word.

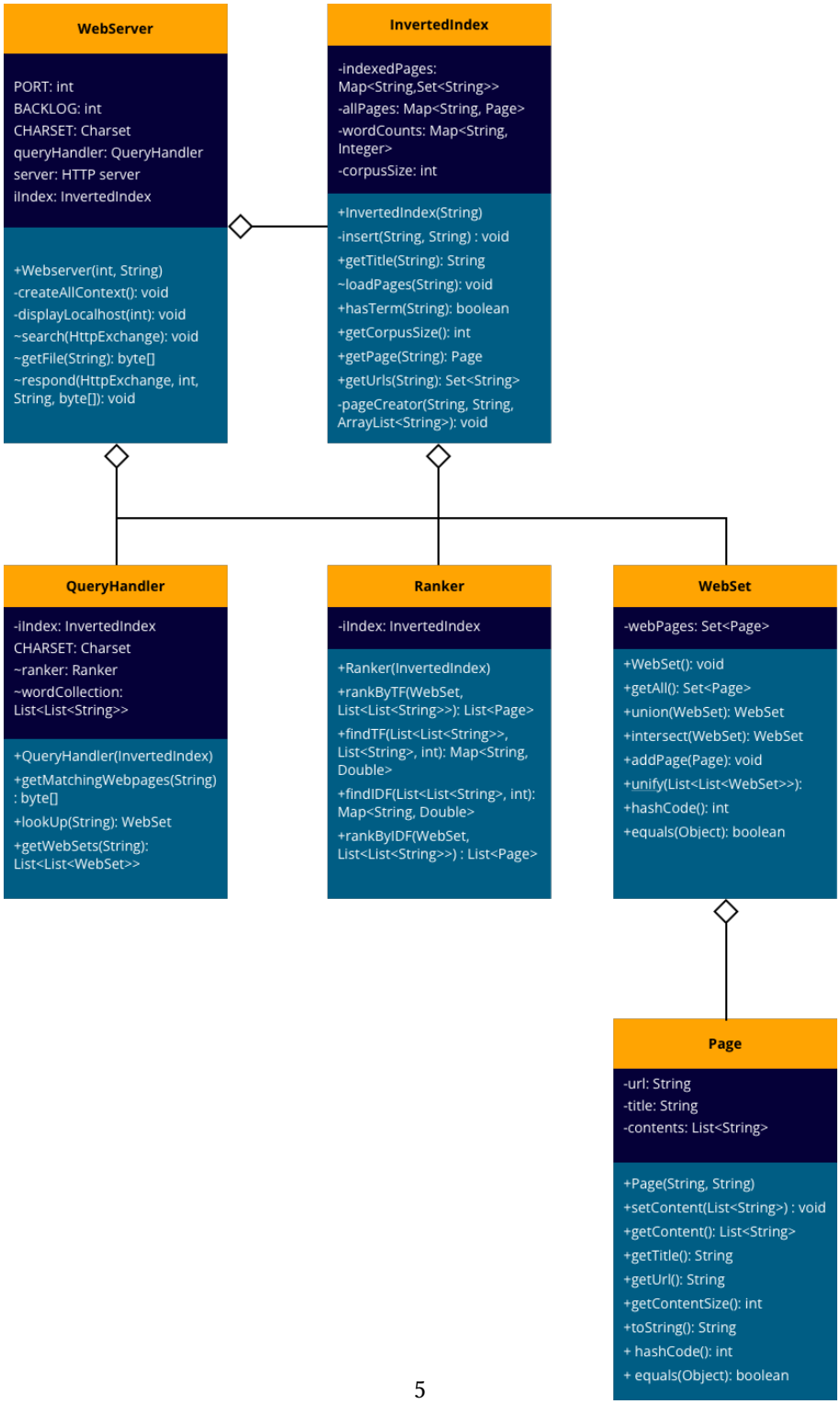

Figure 1: Class Diagram

Approach We have modified a document object with the id attribute "responsesize" in the the code.js to display a message if content of web pages did not match the given query. Furthermore, we have modified the reading of the data files in the `InvertedIndex` class, `loadPages(String filename)` method, such that it only created a web page if it contained a title and and at least one word.

Solution In code.js, we have decided to implement a ternary conditional operator to make our code more concise followed by additional if/else statements. The condition states that if data.length equals to 0 then the program outputs a paragraph containing the following message: "No web page contains the query word." If the condition is not met, then web pages matching the given query will be displayed. We have further modified the style of the document object with the id attribute "urllist" to remove a frame from the url list if no pages are returned. We have implemented the solution using an if/else statement and two properties: style and display. The condition states that if url list does not contain any pages, the element is removed else the program displays search results as a block element. In regards of `loadPages(String filename)` method, first we have identified the following criteria for a page to be valid:

- it must contain an url
- it must contain a title
- it must contain at least one word

In order to check if the given page meets all the criteria, we must track url, title and content. Therefore we have declared three local variables:

- `String currentUrl`
- `String currentTitle`
- `ArrayList<String> currentContents`

Alongside, we have implemented three boolean checks combined with AND operator that checks if these variables are not empty:

- !currentUrl.isEmpty() && !currentContents.isEmpty() && !currentTitle.isEmpty())

First, the program loads contents from the data file and reads them line by line using a for loop. In this for loop, we populate currentUrl, currentTitle and currentContent variables. There are three potential scenarios that we need to be aware of and store the correct data: a line could be a url, a title, or be a part of the content. Therefore we have implemented if/else statements to identify and control the action for each scenario:

- if the line starts with the "*page" prefix - then it must be a url, so the program removes the prefix by using substring() method and assigns the retrieved url as a currentUrl.
- if the condition above fails, then it must be either a title or content. In the data files, we have observed that the title always comes after the page url. Therefore, in order to distinguish between those two, the program checks whether index of the current line equals to the index of url increased by 1. If it's a title, then the program assigns the line as a value of the currentTitle.
- if both conditions above fail, then the line must be a part of the content. So the program adds the line to the list of currentContents.

The program continues to loop through the lines populating currentContent until it reaches the second page. At this point, the currentUrl, currentTitle and currentContents are not empty, meaning the content satisfies all the criteria, so it passes boolean checks allowing for a new page to be created. We have verified the message output through manual testing where we have used a string of random characters that can not be recognized as an actual word and used it as an input in the search engine. Furthermore, we have verified that no valid page is created using unit test based on corpus size variable which we increment by 1 every time a new page is created. They are available in InvertedIndexTest.java.

Reflection As one of the potential shortcomings could be that this solution is based on the assumption that there is no page that starts with an empty space or the data file contains an empty line, which is a correct assumption for all the given datafiles. Furthermore, the data reading logic is based on the pre-defined structure of the content, meaning if page title would come before the url, the solution would not work.

Task 3: Inverted indexes

Task The given task was to make our query algorithm more efficient by implementing a inverted index. An inverted index allows for better performance by sorting and creating a map that matches terms by their document ID. The inversion of the map that occurs allows for searching the occurrence of words within our web pages. The inverted index is created in out `InvertedIndex` class. As mentioned in task 2, this class has a method loadPages() where the creation of the `InvertedIndex` will take place.

Approach An inverted index is created by making a `HashMap` called `title`. Our key for our inverted index is created by mapping a key-value pair of `URL` and `titles`. We also want to store data in a `HashMap`, called `indexedPages` that contains a key value pair of a term, or a line of content and it's `URL`. This information will later be used for our `HashSet` and as such the `URL` value is stored as a `Set`. Therefore, for example, if the content in our page is, "This is a page" and the `URL` is "www.page1.com" for each word in the string it is mapped onto "www.page1.com". Lastly, we have `allPages`. This is a `HashMap` that has a key value pair of the `URL` and page information.

Solution In order to solve this task we needed a way to sort through data files identifying which lines of code are `URL`, `Title`, and `Content`. We have used a scanner to sort through each line of text and three conditions have been set up to sort into variables these different lines as mentioned in task 2, namely, `currentURL`, `currentTitle`, and `currentContent`. The inverted index after being created, is implemented in the `QueryHandler` class as a parameter in the `QueryHandler` constructor. From this point we use the "query" of our users input text to create a `WebSet`. The `WebSet` will be explained in the next section, but the two methods in `QueryHandler` that use the inverted index are `lookUp()` for one word queries and `getWebSets()` for multi-word queries.

Reflections There are potential pitfalls we noticed with the `loadPages()` function and our inverted index, especially with our original implementation, where we put the whole file into an Array List of strings and scanned through them. This implementation was fixed, by using the scanner class, and scanning the text file, so long as their was text to be scanned. Unfortunately, there was still the issue with the enwiki-large.txt file, but this issue is related to hardware requirements in order to scan through the files completely. However for that reason we choose to go back to the readAll. A potential solution to this issue would be to have a database that would be able to maintain these files, or perhaps storing a line number in each page, so that content can be read from the file, and then all the page content, does not have to be stored in RAM.

Task 4: Refined Queries

Task The given task was to enable our search engine to support more complex queries meaning that now we could use multiple search queries instead of a single term.

Approach In order to get a better understanding of the problem of generating sets with relevant web pages that will be displayed to the user, we have used the following mathematical approach:

Suppose the user searches for `Queen`. Then, let W be the set containing all pages, $P_x = \{w \mid w \in W \wedge x \in w\}$ which is the set of pages W in the containing the word x. One might imagine that $P_{Queen} = \{\text{Amalienborg, Sweden, } \dots\}$ `Queen Denmark`
What we now want is pages that contain both "Queen" and "Denmark", these can be denoted as:

$$\begin{aligned} P_{x,y} &= \{w \mid w \in W \wedge (x \in w \wedge y \in w)\} \\ &= P_x \cap P_y \end{aligned}$$

We can again imagine that Amalienborg persists but Sweden is gone

$$P_{Queen,Denmark} = \{\text{Amalienborg}, \dots\}$$

We can generalize this further to

$$P_{x_1,x_2,\dots,x_n} = P_{x_1} \cap P_{x_2} \cap \cdots \cap P_{x_n} = \bigcap_{j=1}^{n} P_{x_j}$$

We can also have queries with sub queries `Queen Denmark OR President USA`
This contains to sub queries Let D be the set of pages to be displayed. Then,

$$D = P_{Queen,Denmark} \cup P_{President,USA}$$

This can be generalized to :

$$\begin{aligned} D &= P_{x_{1,1},x_{1,2},\dots,x_{1,n_1}} \cup P_{x_{2,1},x_{2,2},\dots,x_{2,n_2}} \cup \cdots \cup P_{x_{m,1},x_{2,2},\dots,x_{m,n_m}} \\ &= \bigcap_{j=1}^{n_1} P_{x_{1,j}} \cup \bigcap_{j=1}^{n_2} P_{x_{2,j}} \cup \cdots \cup \bigcap_{j=1}^{n_m} P_{x_{2,m}} \\ &= \bigcup_{i=1}^{m} \left(\bigcap_{j=1}^{n_i} P_{x_{i,j}} \right) \end{aligned}$$

where $x_{i,j}$ is the j'th word of the i' sub query, m is the number of sub queries.

Solution The solution is divided into two separate classes - QueryHandler and WebSet. The QueryHandler class is responsible for processing multiple word or multiple search queries. The WebSet class is responsible for supporting QueryHandler by creating intersections and unions between sets. In the QueryHandler class, we have implemented the following methods to handle the processing of search queries and utilized the Inverted Index:

- `lookUp(String term)` - responsible for processing a single word query, checking if it is contained within loaded pages and returning a WebSet of pages that contain the given search term
- `getMatchingWebpages(String query)` - responsible for retrieving and processing matching pages, and passing them to the WebServer
- `getWebSets (String text)` - responsible for processing the search query, separating it into sub queries and further into single word queries and returning the list of WebSets of pages containing the single word query

In the WebSet class we have developed the following methods to be able to retrieve the intersection of matching queries from multiple sets:

- `union(WebSt in)` - responsible for creating a union of WebSets that contain intersections of matching pages
- `intersect(WebSet in` - responsible for finding an intersection between the sets
- static `unify(List<List<WebSet» webSetList)` - responsible for processing the inputted WebSets and returning the WebSet of unified intersections

First, the program modifies the search input so that it could be utilized when searching for relevant pages using lookUp(String term) method. Also, we had to keep in mind that our data structure in InvertedIndex uses a single search term as a key from the loaded pages, therefore multiple words or multiple queries merged via OR operator had to be split into singular terms. In order to do so, we have implemented a getWebSet() method that takes the search query and applies the following modifications:

- replaces the space encoding %20 with the actual space
- turns the search query into lowercase to make sure there are no inconsistencies between search queries and loaded pages
- splits the multiple queries by " or ", e.g "queen denmark or president usa" would be split into two string sub queries "queen denmark" and "president usa".

Then modified strings are stored into an array list. In order to split sub queries into singular terms, we have implemented a for loop followed by a nested for loop. In the outer for loop, the program creates an array list which will contain the returned WebSets and splits the sub query into singular terms. The reason why we decided to create an array

list is to be able to group the returned sub query WebSets. For example, the search query "queen denmark" will return two WebSets, one for the "queen" and one for the "denmark". If the input is "queen denmark or president usa", we want to keep the Websets of "queen denmark" separately from the returned WebSets of "president usa" as they do not relate to each other. Then "queen denmark" is split into two strings "queen" and "denmark", where the singular term is passed to the inner loop which is responsible for populating a list of list of WebSets, which store all the pages that contain a given term. In order to do so, we call a lookUp(String term) method for every term. As mentioned before, the look up method creates a new WebSet that contains all the pages relevant to the given term. First, it utilizes the InvertedIndex to check whether the given term is contained within the loaded pages by calling containsKey method for the specified key in indexedPages map which contains terms as keys and urls as values. If it passes the check, the program iterates through the set of urls that are mapped to the specified key and one by one retrieves the url. Then this url is used to retrieve a page from the allPages map that contains all the loaded pages. These pages are added into the WebSet. So at this point, we have a WebSet that contains the pages with the key term "queen" and the WebSet containing pages with the key term "denmark". As described above, these two WebSets are stored into an array list which is stored within another list, so once the getWebset method is complete, the return value is the list of list of WebSets. For example, the list of "queen denmark or president usa" would contain: 4 singular query WebSets ("queen", "denmark", "president", "usa") within 2 sub query lists ("queen denmark", "president usa") within 1 search query list ("queen denmark or president usa"). In order to get matching pages for the sub query WebSets, the program utilizes the WebSet object and it's unify method. It returns a new Webset that is populated with all the pages containing "queen" and all the pages containing "denmark". We take in the returned list of list of WebSets and we iterate through it splitting the list into separate lists of WebSets and further into WebSets. Here we have implemented a condition that verifies that a list of WebSets contains at least one page to ensure that the intersection exists. When the program reaches the point where an individual WebSet is extracted, the intersection method is called. In the intersection method, the program returns a new Webset that contains pages that are contained within the given WebSets. First, the program populates webPages set that is instantiated with a WebSet constructor and the WebSet of intersected pages with all the pages containing "queen". Then the program runs the intersection method again, but this time with the WebSet that contains "denmark". Since we do not reset the webPages set, it still contains all the pages previously populated by "queen" WebSet. However, the program creates a new intersection Webset removing preivously stored "queen" pages. Now the program iterates through the WebSet of "denmark" and if the page matches the page that is stored in webPages, that means that this page has both search queries "queen" and "denmark" in their Websets, therefore we populate the newWebSet with the intersected pages. By reseting the intersection Webset and populating webPages with all the input pages, we make sure that we store the pages that only contain both queries, so when the

user searches "queen denmark" or "denmark queen" the results are the same. Then the intersected WebSets are united using the union() method. The first for loop is utilized when there are multiple intersections, for example if the search query contains several sub queries "queen denmark or president usa". Then it iterates through the webpages list that contains pages from the first inputted set and adds these pages into the newWebSet. The second for loop is used to populate the newWebSet with the pages from the next inputted WebSet. If there is only a single intersection, the program stays within the second loop. Finally, the program returns the WebSet of unified intersections to the QueryHandler where the getMatchingWebpages method processes the unified WebSet. It iterates through the WebSet and extracts individual page into an array list which is stored into byte array and passed to a WebServer to return the matching pages to the user. We have tested the solution using unit tests in WebSetTest.java and QueryHandlerTest.java.

Reflection The potential shortcoming is that when given a keyword followed by "or" and then nothing, it reduces search results. For example, if the user inputs "queen", the program returns 281 matches, if the input is "queen or", then the matches are reduced to 78 pages. The program recognizes it as a one sub query and returns matches that contain both "queen" and "or". Also, the program looks for the intersections for one word queries and unifies intersections when there is only one intersection available. Given the additional time, QueryHandler class could potentially be split into two separate classes for processing simple and complex queries so the program does not call intersection and union methods if not applicable.

Task 5: Ranking Algorithms

Task The given task was to implement a ranking algorithm so the search results are ranked by the importance. The task asks to both find term frequency based on a given web pages contents and to find term frequency based on the `InvertedIndex` of the data.

Approach We have decided to look into a term frequency score which would evaluate how relevant the given search query is to a document in a collection of documents. This is done by checking the relevance of a query within a web pages content. The content is matched with the query, and if the query matches a count is taken. Then the frequency of these matches on the query is divided by the total size of the document. Too avoid the pitfall of having words such as "the" ranking as important, we have introduced the inverse-document frequency.

We have followed the mathematical approach described below:

To calculate the TF-IDF value of a term in a given document we have applied the formula: $tfidf(\text{term}, d, D) = tf(\text{term}, d) \cdot idf(\text{term}, D)$ where term is a single word search query, d is the document and D is the corpus of documents.

To calculate the term frequency of the search query within the given document, we have applied the following formula:

$tf(\text{term}, d)$: the number of term within the document divided by the number of all words in the document

To calculate the inverse document frequency of the search query within the corpus of documents, we have applied the following formula:

$idf(\text{term}, D)$: the logarithm of ratio the number of documents in the corpus and the number of documents that contain the term

Solution We have implemented the solution in Ranker.java class. It contains these substantial methods:

- `rankbyTF` - responsible for sorting the pages based of the term frequency
- `rankbyIDF` - responsible for sorting the pages based on the inverse document frequency
- `tfScores` - calculate TF - scores for a single page
- `combineTF` - combines term frequencies from a single page based on query structure
- `rankbyHashedID` - like rank by IDF but implemented with hashmaps and tables, which is faster with many results, it uses the methods below.
- `findTF` - responsible for calculating the term frequency
- `findIDF` - responsible for finding the inverse document frequency

In the `QueryHandler getMatchingWebpages()` method we utilize ranker by calling `rankbyTF`, `rankbyIDF` or the `rankbyHashedID` method.

There are few parts worth mentioning. First, we have implemented a private `RankedPage` class within the class which utilizes the Comparable interface. The reason for this was to use a `compareTo` method to sort the array list alphabetically if tf or idf values were equal, and otherwise sorts by score.

After programming the `rankByTF` we realized, that the `rankByIdf` would be almost identical to the first. Therefore we made an overload of `rankByTf`, that took an extra boolean

	Hashmap findingsOnPage (Page, Hashmap term)		
		Page 1	Page 2
HashMap term (term,tf)	queen	TF	TF
	denmark	TF	TF

Figure 2: Data Structure

argument, `useIdf`. `rankByTF` and `rankByIDF` now just calls this overload with parameter false and true, respectively.

`rankByTF` first makes a list with all words from the query and a another list, specifying the subquery it word on the same index comes from. Then then `webCollectionSet` is looped through, the tf or tfidf are computed for each word and combined using the addition of words in the same query, and using max of scores from two subqueries. That way every page gets a score, and a `rankedPage`, can be inserted to an array. In the end, the array is sorted using the `compareTo` method mentioned before.

Hash-based idf The findTF() method we scan the contents of the page looking for a singular query occurrences within its contents. It calculates the TF value for every search query within the given page by iterating through the list of the given page contents using multiple for loops since we are iterating through two lists: search query list and page contents. The first loop retrieves a single search query and the second for loop retrieves a single word from the page contents. Then, if the query matches the word in the content, the counter of word occurrence is increased by 1. Then we calculate the TF value by dividing the page size with the value of counter. The results are stored within the findings map where search query is a key and TF result is a value and resets the counter.

rankbyTF() method populates a HashMap called findingsOnPage which stores the page as a key and another HashMap called term as a value. The HashMap term stores the search query as a key and the TF result per given page as a value. In order to populate the term HasMap it utilizes findTF() method. Initially, what we were aiming for in this step was to create a data structure like shown in figure 2.

The method utilizes the Entry interface and entry.Set() method. We have decided to use Entry interface and entrySet() method as a workaround to a keySet() method as we kept receiving undefined errors. In this step the program extracts each row of the data structure mentioned above and then retrieves the TF value for each search query from every page. Then the retrieved values are sorted in an ascending order using Collections.sort() method. Sorted pages are then returned back to the QueryHandler.

For the IDF ranking, the program calculates the IDF using findIDF() method. It iterates through the list of search queries and extracts a single word query which is used to retrieve the number of pages that contains it. Here the program calculates the IDF by utilizing Math class and log10() method. We divide the size of the corpus (retrieved in rankbyIDF method) by the number of pages that contain the search query. The results are stored within the findings map where search query is a key and IDF result is a value.

rankbyIDF()

We have tested the solution with unit testx in RankerTest.java.

Reflection It could have beneficial to have spend some more time on testing, and also some, simplification could have been approiate, e.g. minimizing the number of times the idf is computed. We have encountered the bug in our final score results for the hashed-based version. It is not accurate when the search query contains multiple sub queries merged via OR. For example, the search query "queen denmark OR president usa" would return the final score for the entire query instead of calculating for individual sub query "queen denmark" and "president usa". Given the extra time, we would re-design the processing of the search queries to fix the bug.

Conclusion

The improvement of the search engine has been a valuable experience in many perspectives. First of all, while analysing the current and designing the improved architecture for the search engine, we have gained an in-depth knowledge of object-oriented programming and Java. Especially when iterating through the several versions of our class diagram we understood how important it was to plan ahead in order to avoid the situations where the whole team had to go back to the drawing board and re-think the overall architecture from scratch. Furthermore, we have been introduced to a new data structure such like InvertedIndex and gained more experience in the usage of HashMaps and HashSets. We have also learnt about the time term frequency–inverse document frequency which none of us had any experience prior this project. Alongside, we have been shown the importance of unit testing which helped us to discover bugs that we might have easily overlooked during the development process. In general, working on a bigger project and developing code as a team was challenging and yet rewarding experience. We understood that clear communication and usage of correct terminology are the keys to a reduced development time and higher quality of the product. We also became more confident in using git. Also, java docs became a very important tool when trying to read and understand the functionality of the class developed by other team members.